# Representing Memory-Mapped Devices as Objects

Dan Saks
Saks & Associates
www.dansaks.com

 1

## Abstract

Programmers who develop embedded systems often have to assert direct control over hardware resources such as memory-mapped i/o registers. The longstanding practice has been to use concerns over performance as an excuse for writing some pretty nasty code -- heavy in macros, casts and pointer arithmetic. Such code is often hard to get working and hard to maintain. It need not be so.

This talk shows you how to model memory-mapped devices as C++ objects that are more robust, maintainable, and at times, even more efficient than they would otherwise be.

 2

## About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan has written columns for numerous print publications including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. He is currently on leave from writing the online "Programming Pointers" column for *embedded.com*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*. He has also been a Microsoft MVP.

Dan has taught thousands of programmers around the world. He has presented at conferences such as *Software Development* and *Embedded Systems*, and served on the advisory boards for those conferences.

 3

## About Dan Saks

Dan served as secretary of the ANSI and ISO C++ Standards committees and as a member of the ANSI C Standards committee. More recently, he contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. He was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/ Information Science from Case Western Reserve University.

 4

## Common Practice

- You can write very simple declarations to model devices.
- It's less work up front, but…
- …code that accesses devices will be complex and error-prone.

 5

## The Alternative

- You can write more elaborate and accurate declarations to model devices.
- It's more work up front, but…
- …code that accesses devices will be simpler and more robust.

 6

## Pay Now or Pay More Later

- You define each device type at most once.
- You might access each device many times.

 7

## Good General Advice

✓ ***Make interfaces easy to use correctly and hard to use incorrectly.***
— Scott Meyers

✓ ***Program in a style that turns potential run-time errors into compile-time errors.***
— Me

 8

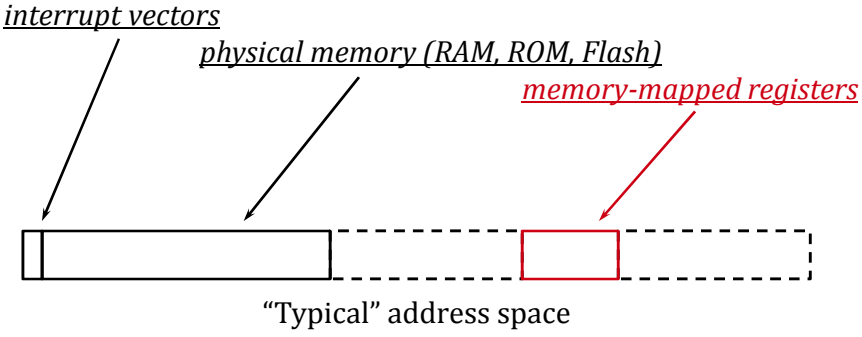
## Device Registers

- Drivers communicate with hardware via ***device registers***.
- Most modern computer architectures use ***memory-mapped addressing***…

 9

## Memory-Mapped Devices

- The architecture disguises the device registers to be addressable like "ordinary" memory:




 10

## Sample Device Registers

- In my example, a UART consists of six device registers:

| Offset | Register | Description |
|---|---|---|
| 0 | ULCON | line control register |
| 4 | UCON | control register |
| 8 | USTAT | status register |
| 12 | UTXBUF | transmit buffer register |
| 16 | URXBUF | receive buffer register |
| 20 | UBRDIV | baud rate divisor register |

 11

## Choosing the Right Integer Type

- Declare each device register using an appropriate integer type.
- For example, a 2-byte register might be a `uint16_t` (from *<cstdint>*).
- Each register in these examples occupies a 4-byte word.
- Using `uint32_t` works well.

 12

## Placing Memory-Mapped Objects

- Normally, you don't choose where objects reside.
- The compiler does, with help from the linker.
- For memory-mapped registers, the compiler doesn't choose.
- The hardware has already chosen.

 13

## Locating Device Registers

- Some compilers let you declare an object at a specified address.
- They provide a non-standard syntax such as:

```
uint32_t UTXBUF0 _at(0x03FFD00C);

uint32_t UTXBUF0 @ 0x03FFD00C;
```

 14

## Locating Device Registers

- Using a macro is common in C:

```
#define UTXBUF0 ((uint32_t *)0x03FFD00C)
```

- In C++, using a const object and "new style" casts is preferable:

```
uint32_t *const UTXBUF0
    = reinterpret_cast<uint32_t *>(0x03FFD00C);
```

- Either way, you can use the pointer to manipulate the register:

```
*UTXBUF0 = c;   // OK: write c's value to UART 0
```

 15

## Locating Device Registers

- Alternatively, you can declare `UTXBUF0` as a reference:

```
uint32_t &UTXBUF0
    = *reinterpret_cast<uint32_t *>(0x03FFD00C);
```

- You must dereference the result of the cast to obtain an object to which the reference can bind.

- Now you can treat `UTXBUF0` as the register itself:

```
UTXBUF0 = c;    // OK: write c's value to UART 0
```

 16

## Modeling Devices

- Many UART operations involve more than one UART register.
- Passing registers separately is error-prone:

```
void UART_put(dev_reg *stat, dev_reg *txbuf, int c);
~~~

UART_put(UTXBUF0, USTAT0, c);    // wrong order

UART_put(USTAT0, UTXBUF1, c);    // mismatching UART #s
```

 17

## Using Structures

- Clustering registers into C structures is better:

```
struct UART {
    dev_reg ULCON;
    dev_reg UCON;
    dev_reg USTAT;
    dev_reg UTXBUF;
    dev_reg URXBUF;
    dev_reg UBRDIV;
};

void UART_put(UART &u, int c);
```

 18

## Using Classes

- Using a class with private members cuts down on improper register accesses:

```
class UART {
public:
    void put(int c);
    ~~~
private:
    dev_reg ULCON;
    dev_reg UCON;
    ~~~
};
```

 19

## Unwelcome Optimizations

- Device registers aren't ordinary memory.
- Device register accesses (reads and writes) may have side effects.
- Compiler optimizations might eliminate register accesses…
- …eliminating those side effects…
- …causing device drivers to fail.

 20

## The Volatile Qualifier

- Declaring an object volatile inhibits optimizations.
- In particular, the compiler can't eliminate accesses to volatile objects…
- …even when it seems safe to do so.

 21

## The Right Dose of Volatility

- This declares `com0` as a "const pointer to a volatile `UART`":

```
UART volatile *const com0
    = reinterpret_cast<UART *>(0x03FFD000);
```

- This declares `com1` as a "reference to a volatile `UART`".

```
UART volatile &com1
    = *reinterpret_cast<UART *>(0x03FFD100);
```

- Here, volatile isn't part of the `UART` type…

 22

## The Right Dose of Volatility

- Failure to declare a UART volatile can lead to a subtle bug:

```
UART volatile &com1              // missing volatile
    = *reinterpret_cast<UART *>(0x03FFD100);
```

- If every UART is volatile, volatile should be part of the UART type…

 23

## The Right Dose of Volatility

- It's unlikely that only some registers are volatile:

```
class UART {
    ~~~
    dev_reg ULCON;
    dev_reg UCON;
    dev_reg volatile USTAT;
    dev_reg volatile UTXBUF;
    dev_reg volatile URXBUF;
    dev_reg UBRDIV;
};
```

 24

## The Right Dose of Volatility

- This doesn't compile:

```
volatile class UART {
    ~~~
};                          // error: missing declarator
```

- The compiler wants to apply `volatile` to a declarator:

```
volatile class UART {   // type UART isn't volatile...
    ~~~
} u;                        // but object u is
```

 25

## The Right Dose of Volatility

- You can use a typedef to define `UART` as a volatile type:

```
typedef volatile class nv_uart {    // either this...
    ~~~
} UART;

typedef class nv_uart {             // ...or this
    ~~~
} volatile UART;
```

- Unfortunately, this leads to other problems…

 26

## The Right Dose of Volatility

- You have to declare all the member functions as volatile:

```
typedef class nv_uart {
public:
    void put(int c) volatile;
    int get() volatile;
    ~~~
private:
    dev_reg ULCON;
    dev_reg UCON;
    ~~~
} volatile UART;
```

 27

## The Right Dose of Volatility

- Typically, all special registers (not just those in UARTs) are volatile.
- In that case, just declare dev_reg as a volatile type:

```
typedef uint32_t volatile dev_reg;
```

- The class definition reverts to its earlier simple form…

 28

## The Right Dose of Volatility

```
class UART {
public:
    void put(int c);
    int get();
    ~~~
private:
    dev_reg ULCON;
    dev_reg UCON;
    ~~~
};
```

- `UART` isn't volatile, but every non-static `UART` data member is.

 29

## The Right Dose of Volatility

- Now you don't need to say `volatile` here:

```
UART *const com0 = reinterpret_cast<UART *>(0x3FFD000);
```

- Or here:

```
UART &com1 = *reinterpret_cast<UART *>(0x3FFD100);
```

 30

## Ensuring Proper Alignment

- Recall that a UART consists of six registers:

| Offset | Register | Description |
|---|---|---|
| 0 | ULCON | line control register |
| 4 | UCON | control register |
| 8 | USTAT | status register |
| 12 | UTXBUF | transmit buffer register |
| 16 | URXBUF | receive buffer register |
| 20 | UBRDIV | baud rate divisor register |

- How can you be sure the `UART` class has this layout?

 31

## Standard-Layout Types

- C++ provides storage layout guarantees, but only for standard-layout types…
- A ***standard-layout type*** is essentially a C type:
  - a scalar type (arithmetic, enumeration, or pointer type)
  - an array with elements of standard-layout type
  - a standard-layout class (possibly declared as a structure or union)…

 32

## Standard-Layout Classes

- A ***standard-layout class*** can have:
  - static and non-static data members, if they're all standard-layout types
  - base classes, if they're all standard-layout types
  - static and non-virtual member functions
  - nested constants and types

 33

## Standard-Layout Classes

- A standard-layout class ***can't*** have virtual functions or virtual base classes:

```
class timer {
public:
    ~~~
    void enable();
    virtual value_type get();   // not standard layout
    ~~~
};
```

 34

## Standard-Layout Classes

- All non-static data members of a standard-layout class must have the same access control.

```
class widget {
public:
    dev_reg status;
protected:                          // not standard layout
    dev_reg control;
    dev_reg data;
};
```

 35

## Standard-Layout Classes

- All non-static data members of a standard-layout class must be declared in the most derived class or in the same base class:

```
struct IOP {
    dev_reg IOPMOD;
    dev_reg IOPCON;          // standard-layout
};

class switches: public IOP {
public:
    dev_reg IOPDATA;         // not standard-layout
};
```

 36

## No Need to Guess

- You can use a static assertion with a type trait to check:

```
#include <type_traits>

class timer {
    ~~~
};

static_assert(
    is_standard_layout<timer>::value,
   "timer isn't standard layout"
);
```

 37

## Layout Guarantees

- For standard-layout classes, C++ guarantees only that:
  - The first non-static data member is at offset zero.
  - Every other non-static data member has an offset greater than the data member declared just before it.

 38

## Padding

- A class may have padding bytes after any non-static data member...
- ...but not before the first.
- `UART` is a standard-layout class.
- `ULCON` will be at offset zero within `UART`.
- What about the offset of other data members?

 39

## Packing

- Some compilers offer pragmas to control packing, as in:

```
#pragma pack(push, 4)
class UART {
    ~~~
};
#pragma pack(pop)
```

- Some GNU C++ dialects support type attributes such as:

```
class UART __attribute__(packed) {
    ~~~
};
```

 40

## Sooner Rather Than Later

- Misaligned members in device classes often lead to runtime failures.
- `static_assert` can catch misaligned members at compile time:

```
class UART {
    ~~~
};

static_assert(offsetof(UART, UCON) == 4, "~~~");
static_assert(offsetof(UART, ULCON) == 8, "~~~");
```

 41

## Using Static Assertions to Enforce Layout

- `offsetof(t, m)` (defined in <*cstddef*>) returns the offset in bytes of member `m` from the beginning of class type `t`.
- If `t` isn't a standard-layout class, the behavior is undefined.
- Checking the offset of each member can be tedious.
- This might be all you need:

```
static_assert(sizeof(UART) == 6 * sizeof(dev_reg));
```

 42

## Nested Types and Constants

- Some UART member functions have parameter types specific to the class.
- These types should be public class members:

```
class UART {
public:
    enum baud_rate {
        BR_9600 = 162 << 4, BR_19200 = 80 << 4, ~~~
    };
    ~~~~
}
```

 43

## Using a Constructor

- UART objects should be initialized before use.
- A constructor is the way to go:

```
class UART {
public:
    UART(baud_rate br = BR_9600) {
        disable();
        set_speed(br);
        enable();
    }
    ~~~~
};
```

 44

## Constructors

- Constructors provide ***guaranteed initialization***:
  - If class `UART` has a constructor, the compiler "guarantees" to initialize every `UART` object by calling a constructor.
- Unfortunately, a cast can invalidate the guarantee…

 45

## Constructors

- Memory-mapped objects aren't "normal" objects in that:
  - You don't define any objects of the `UART` type.
  - You just set up pointers or references to existing locations using `reinterpret_cast`.
- The compiler fails to generate a constructor call automatically…

 46

## Constructors

- Recall the definition for the UART "object":

```
UART &com0 = *reinterpret_cast<UART *>(0x3FFD000);
```

- The cast invalidates the initialization guarantee…
- The declaration locates the UART object, but doesn't initialize it.
- Fortunately, you can construct the UART object by using a placement new-expression…

 47

## Constructors and New-Expressions

- A ***new-expression*** allocates memory by calling an operator new.
- C++ provides a default implementation for a global operator new, declared in standard header *<new>* as:

```
void *operator new(size_t n);
```

- Parameter n represents the size (in bytes) of the requested storage.

 48

## Constructors and New-Expressions

- A new-expression has the form:

```
p = new T (v);  // (v) is optional
```

- It translates into something (sort of) like:

```
p = static_cast<T *>(operator new(sizeof(T)));
p->T(v);        // constructor "call" (not real C++)
```

- `p->T(v)` is my notation for "apply to `*p` the `T` constructor that accepts argument `v`".

 49

## Constructors and Placement New

- C++ provides a version of `operator new` that you can use to "place" an object at a specified location:

```
void *operator new(size_t, void *p) noexcept {
    return p;
}
```

- It ignores its first parameter and simply returns its second.

 50

## Constructors and Placement New

- A ***placement new-expression*** has the form:

```
p = new (region) T (v);      // (v) is optional
```

- It translates into something along the lines of:

```
p = static_cast<T *>(operator new(sizeof(T), region));
p->T(v);
```

- It constructs a `T` object in the storage addressed by `region`.

 51

## Constructors and Placement New

- You can use placement new to invoke the `UART` constructor:

```
UART *const com0 = reinterpret_cast<UART *>(0x3FFD000);
com0 = new (com0) UART;
```

- Assigning the new-expression to `com0` isn't necessary:

```
UART *const com0 = reinterpret_cast<UART *>(0x3FFD000);
new (com0) UART;
```

 52

## Constructors and Placement New

- You can fold both statements into a single one:

```
UART *const com0
    = *new (reinterpret_cast<UART *>(0x3FFD000)) UART;
```

- That might not be an improvement.
- You can use a reference instead of a pointer:

```
UART &com0 = *reinterpret_cast<UART *>(0x3FFD000);
new (&com0) UART;
```

 53

## Constructors and Placement New

- If the constructor accepts arguments, placement new will let you pass them:

```
UART &com0 = *reinterpret_cast<UART *>(0x3FFD000);
new (&com0) UART (UART::BR_19200);
```

 54

## Class-Specific New

- C++ lets you declare `operator new` as a class member.
- If `T` is a class with a member `operator new`, then this uses `T`'s `operator new`:

```
p = new T (v);  // (v) is optional
```

- A member `operator new` is a static member, even if not declared so explicitly.

 55

## Class-Specific New

- A member `operator new` can place a device at a specified memory-mapped address:

```
class UART {
public:
    void *operator new(size_t) {
        return reinterpret_cast<void *>(0x3FFD000);
    }
    ~~~
};
```

 56

## Class-Specific New

- Now, you can create a UART object using a familiar-looking new-expression:

```
UART *const com0 = new UART;
```

- It uses the UART's operator new to "place" the UART object in its memory-mapped location.
- It uses the UART's default constructor to initialize the object.
- Cool.

 57

## Class-Specific New

- Alternatively, you can bind a reference to the "allocated" UART:

```
UART &com0 = *new UART;
```

- It's an unusual-looking new-expression, but…
- …it makes com0 look like a UART, not a "pointer to UART":

```
com0.put(c);
```

 58

## What About The Other UARTs?

- The hardware supports four UARTs.
- UART's `operator new` supports only one:

```
class UART {
public:
    void *operator new(size_t) {
        return reinterpret_cast<void *, 0x3FFD000);
    }
    ~~~
};
```

 59

## Member Placement New

- You can augment `operator new` with additional parameters.
- Here, the additional parament specifies the UART number:

```
class UART {
public:
    void *operator new(size_t, int n) {
        return reinterpret_cast<void *>(
            0x3FFD000 + n * 0x1000
        );
    ~~~
};
```

 60

## Member Placement New

- Using this `operator new`, you can write:

```
UART &com0 = *new (0) UART;      // use UART 0
```

- Using a different UART is easy:

```
UART &com2 = *new (2) UART;      // use UART 2
```

- This works correctly only when the placement argument is 0 through 3, inclusive…

 61

## Preventing Errors

- Unfortunately, it compiles for other values:

```
UART &com2 = new (42) UART;      // compiles, but fails
```

- You can't prevent this with a static assertion.
- You could use a run-time check to restrict the placement argument.
- Even better, you can use an enumeration type…

 62

## Preventing Errors

```
class UART {
public:
    enum uart_number { zero, one, two, three };
    void *operator new(size_t, uart_number n) {
        return reinterpret_cast<void *>(
            0x3FFD00 + n * 0x1000);
    }
    ~~~
};
```

 63

## Yet Another Way

- Using this `operator new`, you can write:

```
UART &com0 = new (UART::zero) UART;
```

- Now your choice of UART number is limited to only `zero` (= 0) through `three` (= 3).

- Now, this won't compile:

```
UART &com2 = new (42) UART;
```

- Good Thing.

 64

## Summary

- ✓ ***Write data declarations that model the hardware as precisely as possible.***
- ✓ ***If you do it well, writing code to manipulate the hardware will be much easier.***